Deep learning workshop

Homework

A lot more practicing with tensors. I advice you to look up the functions ‘permute’ and ‘cat’ again before starting :)

Goals: Learn to work with autoencoders. Experience how convolutional layers with kernel size of 1 can be useful. See how datasets can be automatically generated. Experiment a bit with manual dataset creation. Learn how to use train validation and test sets. Gain insight why training using cross entropy loss is faster than using the mean squared error loss.

Assignment 1: The big autoencoder assignment!

Introduction:
During the practical of week 7, you already saw autoencoders trained on the MNIST handwritten dataset using a simple linear variational autoencoder.
Here we are going to experiment on a more fun and complex (yet almost as small (32* 32 pixels)) dataset of colored images using a convolutional variational autoencoder. Don’t worry, the difficult part is already pre-programmed and can be found in the dropbox folder with the name: ‘**vae_example_conv.py**’.

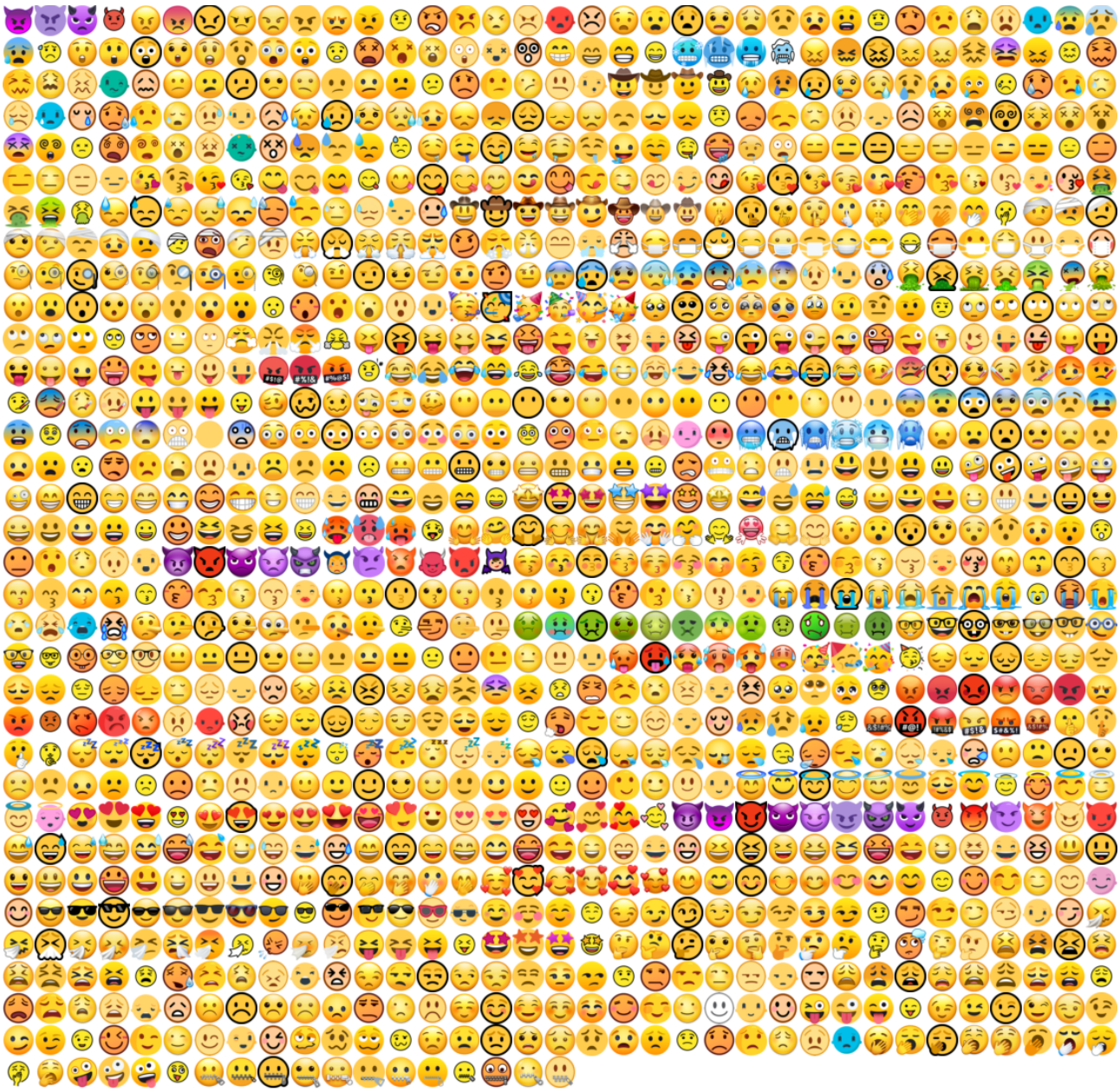
The dataset:
Instead of restricting yourself to pre-assembled datasets, you can easily create your own datasets using ‘webscraping’. This is a technique used to easily extract data from websites. For this assignment I used webscraping to create a datset of ‘emoji’s’, the friendly yellow faces used in social media. (mined from the website: 'https://emojipedia.org’)
To show that webscraping can be done with only a few lines of code (and no external python libraries), I added the script to the dropbox folder (**webscraper.py**).

Assignment 1.1 data preprocessing:
The whole dataset can be seen on the next page, and downloaded from the dropbox folder with the name: ‘**dataset.png**’.
Write a function that extracts all the individual emoji’s and combine them in a single torch tensor. Use the following pseudo-code:

```
def image2dataset(imageLocation, outputShape):
    if outputShape == 'convolutional':
        return tensor_with_shape -> (nmbr_emojis, 3, 32, 32)
    if outputShape == 'linear':
        return tensor_with_shape -> (nmbr_emojis, 3072)
```

Since we are going to use binary cross entropy, make sure the values are between 0 and 1.

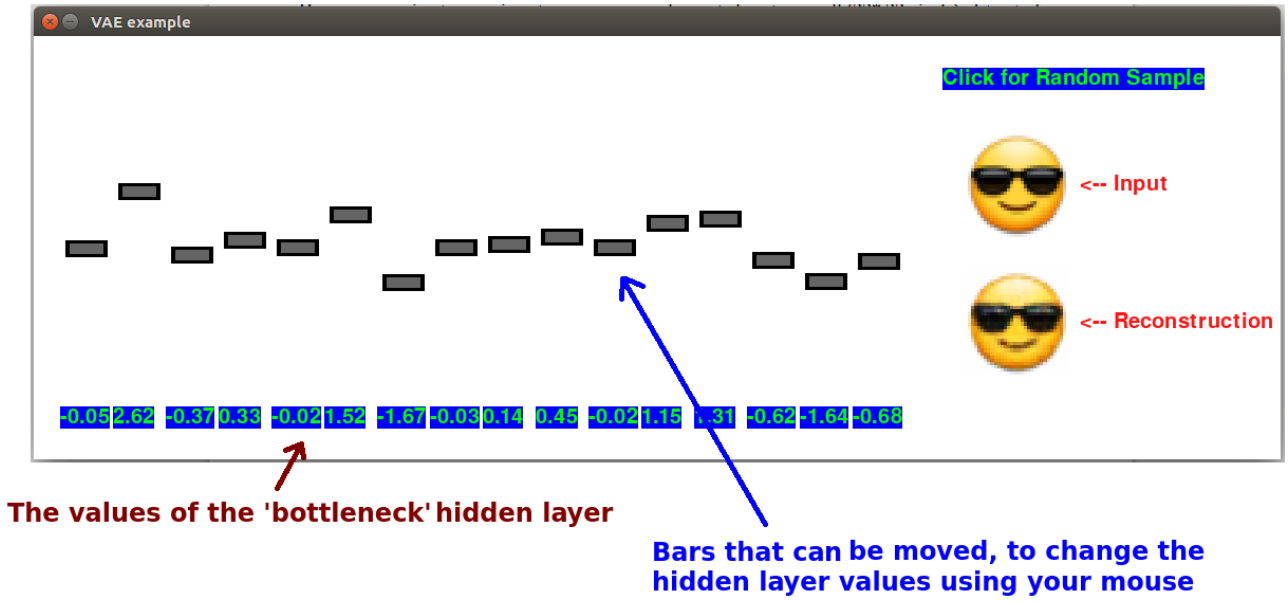
tip: install the ‘**imageio**’ python library using pip. Which has the function ‘**imageio.imread**’ to load an image into python. The previous homework assignments contain tips how to swap axes.



Assignment 1.2, inner scientist: Lets explore what latent variables the autoencoder has learned. If done well, it should have learned **independent features** that can describe an input image.
To prevent wasting a day of training the network yourself, and make directly comparing results with your colleagues possible, I have pre-trained a model which you can download from the dropbox folder (**checkpoint.pth.tar**). Use this code to load the pretrained model:

```
net = VAE()
checkpoint = torch.load('checkpoint.pth.tar')
net.load_state_dict(checkpoint['state_dict'])
```

Also I created a gui (graphical user interface) to easily play around with the latent variables. A screenshot of the program can be seen below. To use the gui install the libraries '**imageio**' and '**pygame**'. Just make sure the '**gui.py'** file is in the same folder as '**vae_example_conv.py**' file.
Explore what each of the 16 latent variables represent. Do this in a systematic manner by adjusting only one variable at the time on different input images. Try to see if attributes like shape, drawing_style or color remain the same when adjusting the values.



Assignment 1.3 latent space vs pixel space: The decoder part of the variational autoencoder has learned to construct an image using 16 variables. Since the learned features are ideally independent from each other, a latent interpolation (combination) of two images should also return a 'plausible' image. This property should not be expected when interpolating two images in pixel space.
To practice your pytorch programming skills, create a function that can interpolate between two input images, both in pixel as in latent space. An example can be seen below. Save one or more funny results. (saving an image can be done using imageio.imsave(filename, numpy_matrix)

Assignment 1.4 math with latent vectors.
In week 7 we saw that you could do 'math' with latent vector representations.
Two examples were given :

'**hamburger**' – '**cow**' + '**pig**' → '**hotdog**'.

'**antibiotics**' - '**bacteria**' + '**cancer**' → '**chemotherapy**'.

Lets do the same with our emoji's. I want you to do the following:

a b c

d

e

Use the latent representations of the images for the math part, and save the reconstructions of the resulting latent vector.
The images can be downloaded from the dropbox folder (**a.png**, **b.png**, **c.png**, **d.png** and **e.png**).

Assignment 1.5 Anomaly detection.

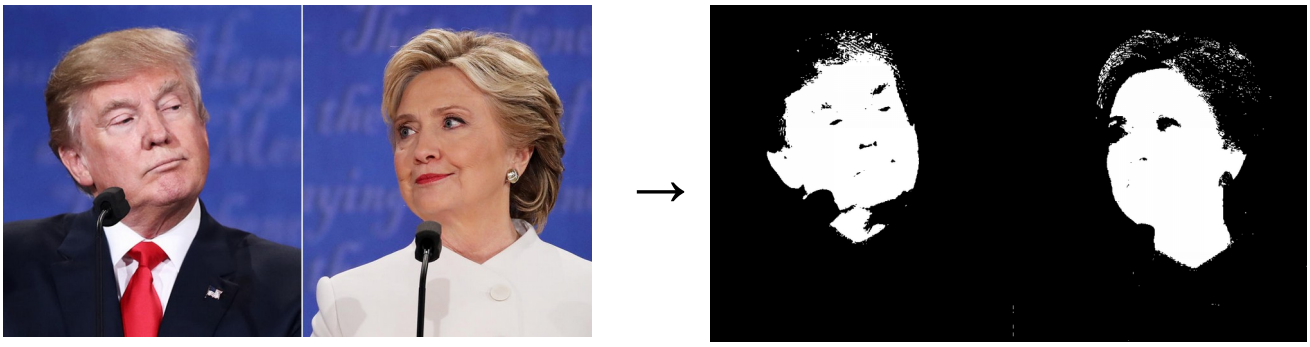
method 1 **Step 1:** Use the reconstruction error of the autoencoder to find the 10 best reconstructed images in the dataset and the 10 worst reconstructions. (use the VAE 'test' function from vae_example_conv.py) Save these resulting images from the dataset as a single image of 10 x 2 emojis.

method 2 **Step 2:** Calculate the 'average' emoji (in pixel space, not latent space). Also, calculate the euclidean distance between all emojis and this average emoji. Save the 10 lowest distance emojis and the 10 highest distance emojis as a single image.

Question: Step 1 and step 2 are both methods to detect outliers/anomalies. However they disagree on which emojis are the main outliers. For each method, what do you think is the main feature it uses to distinguish between normal and anomaly?

Assignment 2. Skin detection

Introduction:
In this assignment you are going to build your own train, evaluation and test datasets. And also experiment with convolutional layers with a kernel size of 1.
The problem: To do tasks like face detection, image segmentation or hand gesture detection, it can be very useful to know which pixels of an image correspond to human skin. Therefore skin detection as a preprocessing technique could significantly help in these tasks. To make the technique as general as possible, only the Red-Green-Blue pixel values are used for predicting the probability that a pixel belongs to human skin. An example of binary skin detection can be seen below:



Datasets:
You might think this is going to be a lot of work, the opposite is true. Deep learning is ideal for lazy people. With other machine learning algorithms you need to label each pixel of an image into the categories skin or non-skin. With deep learning you just need the label the complete photo if there is skin in it or not, and the algorithm will figure out where on its own.

Create three datasets:
- Train set → 10 to 20 images of photos containing human skin. Think of faces, hands, arms, backs etc. This will be your positive set.
  Also 10 to 20 photos without human skin. Think of clothing, nature, walls etc. This will be your negative set.
- Validation set → 3 different photos with skin somewhere in it.These will be used to see how well the neural network performs. The network will not be trained on these images.
- Test set → Another set of photos (3 to 5?) with human skin somewhere in it. These photos will be used to see how well the network will generalize to new data.

Make sure your images are not too large. I, for example, resized my images to 100 x 100 pixels to make batch optimization possible. But you can decide for yourself how you want to train your network.

Neural architecture: I suggest to use the following pseudo-code:

```
model = nn.Sequential(  nn.Conv2d(in_channels=3, out_channels=?, kernel_size=1),
                        activation_function,
                        nn.Conv2d(in_channels=?, out_channels=1, kernel_size=1)
                        nn.Sigmoid()
                      )
```

But you are allowed to make your own modifications of course...

Training:
Each image has a single label (0 or 1) based on if it contains skin or not. Therefore during training the neural output should also be reduced to a single value. Here we will do that by calculating the average value of the output per image.

Steps, just as seen many times before:

for all images → feed to network → retrieve output → calculate the average → calculate the loss given this 'average value' and the 'true label' → calculate gradients → apply gradients → clear gradients from network → repeat

For the loss function I advice to use 'F.binary_cross_entropy(prediction, target)'. You can also use mean squared error, but the optimization will take many more steps.

After training (you decide when to stop), see how well (visually inspect) the neural network performs on your validation images. Do this by saving the output from the network as images. Add new photos to your training set to correct for mistakes the network makes and retrain it. (never train on images from your validation set directly!!!)

When you are happy with your results on the validation set, see how well it performs on the final test set. You are not allowed to retrain your network based on these results anymore :)

Good luck!

Binary cross entropy insight. (not an asignment)

During the previous assignments you already used binary cross entropy to define your loss. This function significantly reduces the number of epochs needed to train your network compared to other error/loss functions such as ‘mean squared error’. To illustrate why, look at the image below. Here the two previously mentioned functions are compared with each other. What you see are the losses of all possible values between 0 and 1 and the ‘true label=0.3’. For binary cross entropy: notice that when predictions are far from the true label, the loss has large slopes and near the true label small slopes. Since we are using gradient descent as the optimizing algorithm, the slope also determines the gradient size/magnitude.

Large gradients → large steps toward correct solution
Small gradients → small steps toward correct solution

If you look at the mean squared error loss function, the slope changes less drastically when moving away from the ‘true label’. This means it takes smaller steps (has smaller gradients) relative to binary cross entropy. Therefore, necessarily it must take more steps to find the same local minimum.

Binary cross entropy can only ‘know’ if a prediction is a little bit wrong or very wrong because it assumes all values are between 0 and 1. Mean squared error does not have this assumption and can therefore be used more generally (for example on large or negative values) at the cost of slower optimization.

In conclusion, binary cross entropy works well because it will take large steps when the prediction is very wrong, and small steps if the prediction is almost right.